Clocks as Types
in Synchronous Dataflow Languages

Marc Pouzet
LRI & INRIA
Univ. Paris-Sud 11
Orsay

IFIP WG 2.8 – 9/06/2009

(joint work with Albert Cohen, Louis Mandel, Florence Plateau)
Synchronous Dataflow Languages

Model/program critical embedded software.

The idea of Lustre:

- directly write stream equations as executable specifications
- provide a compiler and associated analyzing tools to generate embedded code

E.g, the linear filter:

\[ Y_0 = bX_0, \forall n \; Y_{n+1} = aY_n + bX_{n+1} \]

is programmed by writing, e.g:

\[ Y = (0 \rightarrow a \cdot \text{pre}(Y)) + Z; \]
\[ Z = b \cdot X \]

we write invariants

- other primitives to deal with slow and fast processes (sub/over-sampling);
  not necessarily periodic
An example of a SCADE sheet
Dataflow Semantics

Kahn Principle: The semantics of process networks communicating through unbounded FIFOs (e.g., Unix pipe, sockets)?

- message communication into FIFOs (send/wait)
- reliable channels, bounded communication delay
- blocking wait on a channel. The following program is forbidden
  
  if (A is present) or (B is present) then ... 

- a process = a continuous function \((V^\infty)^n \rightarrow (V'^\infty)^m\).

Lustre:
- Lustre has a Kahn semantics (no test of absence)
- A dedicated type system (clock calculus) to guaranty the existence of an execution with no buffer (no synchronization)
Pros and Cons of KPN

(+): **Simple semantics**: a process defines a function (determinism); composition is function composition

(+): **Modularity**: a network is a continuous function

(+): **Asynchronous distributed execution**: easy; no centralized scheduler

(+/-): **Time invariance**: no explicit timing; but impossible to state that two events happen at the same time.

\[
\begin{array}{ccccccc}
  x & = & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \cdots \\
  f(x) & = & y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \cdots \\
  f(x) & = & y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \cdots \\
\end{array}
\]

This appeared to be a useful model for video apps (TV boxes): Sally (Philips NatLabs), StreamIt (MIT), Xstream (ST-micro) with various “synchronous” restriction \textit{à la SDF} (Edward Lee)
A small dataflow kernel

A small kernel with minimal primitives

\[ e ::= e \mathsf{fby} e \mid \mathit{op}(e, \ldots, e) \mid x \mid i \]
\[ \quad \mid \mathit{merge} e e e \mid e \mathit{when} e \]
\[ \quad \mid \lambda x.e \mid e e \mid \mathit{rec} x.e \]

\[ \mathit{op} ::= + \mid - \mid \mathit{not} \mid \ldots \]

- function (\(\lambda x.e\)), application (\(e e\)), fix-point (\(\mathit{rec} x.e\))
- constants \(i\) and variables (\(x\))
- dataflow primitives: \(x \mathsf{fby} y\) is the unitary delay; \(\mathit{op}(e_1, \ldots, e_n)\) the point-wise application; sub-sampling/oversampling (\(\mathit{when}/\mathit{merge}\)).
**Dataflow Primitives**

<table>
<thead>
<tr>
<th></th>
<th>$x$</th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>$y_0$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$y_3$</td>
<td>$y_4$</td>
<td>$y_5$</td>
<td></td>
</tr>
<tr>
<td>$x + y$</td>
<td>$x_0 + y_0$</td>
<td>$x_1 + y_1$</td>
<td>$x_2 + y_2$</td>
<td>$x_3 + y_3$</td>
<td>$x_4 + y_4$</td>
<td>$x_5 + y_5$</td>
<td></td>
</tr>
<tr>
<td>$x \ fby \ y$</td>
<td>$x_0$</td>
<td>$y_0$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$y_3$</td>
<td>$y_4$</td>
<td></td>
</tr>
<tr>
<td>$h$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$x' = x \ when \ h$</td>
<td>$x_0$</td>
<td>$x_2$</td>
<td>$x_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$z$</td>
<td>$z_0$</td>
<td>$z_1$</td>
<td>$z_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge $h \ x' \ z$</td>
<td>$x_0$</td>
<td>$z_0$</td>
<td>$x_2$</td>
<td>$z_1$</td>
<td>$x_4$</td>
<td>$z_2$</td>
<td></td>
</tr>
</tbody>
</table>

**Sampling:**

- if $h$ is a boolean sequence, $x \ when \ h$ produces a sub-sequence of $x$
- merge $h \ x \ z$ combines two sub-sequences
Kahn Semantics

Every operator is interpreted as a stream function \((V^\infty = V^* + V^\omega)\). E.g., if \(x \mapsto s_1\) and \(y \mapsto s_2\) then the value of \(x + y\) is \(+^\#(s_1, s_2)\)

\[
\begin{align*}
i# &= i.i#
+^\#(x.s_1, y.s_2) &= (x + y).+^\#(s_1, s_2)
(x.s_1)fby^#s_2 &= x.s_2
x.s\when^#1.c &= x.(s\when^#c)
x.s\when^#0.c &= s\when^#c
\text{merge}^#1.c\ x.s_1\ s_2 &= x.\text{merge}^#c\ s_1\ s_2
\text{merge}^#0.c\ s_1\ y.s_2 &= y.\text{merge}^#c\ s_1\ s_2
\end{align*}
\]
Some programs generate monsters.

If \( x = (x_i)_{i \in \mathbb{N}} \) then \( \text{even}(x) = (x_{2i})_{i \in \mathbb{N}} \) and \( x \& \text{even}(x) = (x_i \& x_{2i})_{i \in \mathbb{N}} \).

**Unbounded FIFOs!**

- must be rejected statically
- every operator is finite memory through the composition is not: all the complexity (synchronization) is hidden in communication channels
- the Kahn semantics does not model time, i.e., impossible to state that two event arrive **at the same time**
**Synchronous (Clocked) streams**

Complete streams with an explicit representation of absence ($\text{abs}$).

\[ x : (V^{\text{abs}})_{\infty} \]

**Clock**: the clock of $x$ is a boolean sequence

\[ IB = \{0, 1\} \]

\[ \text{CLOCK} = IB^\infty \]

\[ \text{clock}\ \epsilon = \epsilon \]

\[ \text{clock}\ (\text{abs}.x) = 0.\text{clock}\ x \]

\[ \text{clock}\ (v.x) = 1.\text{clock}\ x \]

**Synchronous streams**:

\[ ClStream(V, cl) = \{ s / s \in (V^{\text{abs}})_{\infty} \land \text{clock}\ s \leq_{\text{prefix}} cl \} \]

**An other possible encoding**: $x : (V \times \mathbb{N})_{\infty}$
Dataflow Primitives

Constant:

\[
\begin{align*}
i#(\epsilon) &= \epsilon \\
i#(1.cl) &= i.i#(cl) \\
i#(0.cl) &= \text{abs}.i#(cl)
\end{align*}
\]

Point-wise application:

Synchronous arguments must be constant, i.e., having the same clock

\[
\begin{align*}
+\#(s_1, s_2) &= \epsilon \text{ if } s_i = \epsilon \\
+\#(\text{abs}.s_1, \text{abs}.s_2) &= \text{abs}.+\#(s_1, s_2) \\
+\#(v_1.s_1, v_2.s_2) &= (v_1 + v_2).+\#(s_1, s_2)
\end{align*}
\]
Partial definitions

What happens when one element is present and the other is absent?

Constraint their domain:

\((+) : \forall cl : CLOCK. ClStream(int, cl) \times ClStream(int, cl) \rightarrow ClStream(int, cl)\)

i.e., \((+)\) expect its two input stream to be on the same clock \(cl\) and produce an output on the same clock.

These extra conditions are **types** which must be statically verified.

**Remark (notation):** Regular types and clock types can be written separately:

- \((+) : int \times int \rightarrow int \quad \leftarrow\text{its type}\)
- \((+) :: \forall cl.cl \times cl \rightarrow cl \quad \leftarrow\text{its clock type}\)

In the following, we only consider the clock type.
Sampling

\begin{align*}
  s_1 \text{ when} & \# s_2 = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
  (\text{abs}.s) \text{ when} & \# (\text{abs}.c) = \text{abs}.s \text{ when} \# c \\
  (v.s) \text{ when} & \# (1.c) = v.s \text{ when} \# c \\
  (v.s) \text{ when} & \# (0.c) = \text{abs}.x \text{ when} \# c \\

  \text{merge } c s_1 s_2 = \epsilon \text{ if one of the } s_i = \epsilon \\
  \text{merge } (\text{abs}.c)(\text{abs}.s_1)(\text{abs}.s_2) = \text{abs}.\text{merge } c s_1 s_2 \\
  \text{merge } (1.c)(v.s_1)(\text{abs}.s_2) = v.\text{merge } c s_1 s_2 \\
  \text{merge } (0.c)(\text{abs}.s_1)(v.s_2) = v.\text{merge } c s_1 s_2
\end{align*}
### Examples

<table>
<thead>
<tr>
<th>base = (1)</th>
<th>1 1 1 1 1 1 1 1 1 1 1 1 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9 \ x_{10} \ x_{11} \ ...$</td>
</tr>
<tr>
<td>$h = (10)$</td>
<td>1 0 1 0 1 0 1 0 1 0 1 0 ...</td>
</tr>
<tr>
<td>$y = x$ when $h$</td>
<td>$x_0 \ x_2 \ x_4 \ x_6 \ x_8 \ x_{10} \ x_{11} \ ...$</td>
</tr>
<tr>
<td>$h' = (100)$</td>
<td>1 0 0 1 0 0 1 0 0 1 ...</td>
</tr>
<tr>
<td>$z = y$ when $h'$</td>
<td>$x_0 \ x_6 \ x_{11} \ ...$</td>
</tr>
<tr>
<td>$k$</td>
<td>$k_0 \ k_1 \ k_2 \ k_3 \ ...$</td>
</tr>
<tr>
<td>merge $h' \ z \ k$</td>
<td>$x_0 \ k_0 \ k_1 \ x_6 \ k_2 \ k_3 \ ...$</td>
</tr>
</tbody>
</table>

let clock five =
  let rec f = true fby false fby false fby false fby false fby f in f
let node stutter x = o where
  rec o = merge five x ((0 fby o) whennot five) in o

**stutter(nat)** = 0.0.0.0.1.1.1.1.2.2.2.2.3.3...
Sampling and clocks

- \( x \text{ when}^\# y \) is defined when \( x \) and \( y \) have the same clock \( cl \)
- the clock of \( x \text{ when}^\# c \) is written \( cl \text{ on } c \) : “\( c \) moves at the pace of \( cl \)”

\[
\begin{align*}
s \text{ on } c &= \epsilon \text{ if } s = \epsilon \text{ or } c = \epsilon \\
(1.cl) \text{ on } (1.c) &= 1.cl \text{ on } c \\
(1.cl) \text{ on } (0.c) &= 0.cl \text{ on } c \\
(0.cl) \text{ on } (\text{abs}.c) &= 0.cl \text{ on } c
\end{align*}
\]

We get:

- \( \text{when} : \forall cl. \forall x : cl. \forall c : cl. cl \text{ on } c \)
- \( \text{merge} : \forall cl. \forall c : cl. \forall x : cl \text{ on } c. \forall y : cl \text{ on } \text{not } c. cl \)

Written instead:

- \( \text{when} : \forall cl. cl \rightarrow (c : cl) \rightarrow cl \text{ on } c \)
- \( \text{merge} : \forall cl. (c : cl) \rightarrow cl \text{ on } c \rightarrow cl \text{ on } \text{not } c \rightarrow cl \)
Checking Synchrony

The previous program is now rejected.

This is a now a **typing error**

```haskell
let even x = x when half
let non_synchronous x = x & (even x)
```

This expression has clock 'a on half, but is used with clock 'a

**Final remarks:**
- We only considered **clock equality**, i.e., "two streams are either synchronous or not"
- Clocks are used extensively to generate **efficient sequential code**
From Synchrony to Relaxed Synchrony

- can we compose non strictly synchronous streams provided their clocks are closed from each other?

- communication between systems which are “almost” synchronous

- model jittering, bounded delays

- Give more freedom to the compiler, generate more efficient code, translate into regular synchronous code if necessary
Incrustation of a Standard Definition (SD) image in a High Definition (HD) one

- **downscaler**: reduction of an HD image (1920×1080 pixels) to an SD image (720×480 pixels)
- **when**: removal of a part of an HD image
- **merge**: incrustation of an SD image in an HD image

**Question**:
- buffer size needed between the **downscaler** and the **merge** nodes?
- delay introduced by the picture in picture in the video processing chain?
Too restrictive for video applications

- streams should be synchronous
- adding buffer (by hand) difficult and error-prone
- compute it automatically and generate synchronous code

relax the associated clocking rules
- based on the use of *infinite ultimately periodic sequences*
- a precedence relation $c_{l_1} <: c_{l_2}$
Ultimately periodic sequences

\( \mathbb{Q}_2 \) for the set of infinite periodic binary words.

\[
\begin{align*}
(01) &= 01 01 01 01 01 01 01 01 \ldots \\
0(1101) &= 0 1101 1101 1101 1101 1101 1101 1101 \ldots \\
\end{align*}
\]

- 1 for presence
- 0 for absence

Definition:

\[
w := u(v) \quad \text{where } u \in (0 + 1)^* \text{ and } v \in (0 + 1)^+
\]

WG2.8 meeting
Clocks and infinite binary words

\[ w_1 \]

\[ \mathcal{O}_w(i) = \text{cumulative function of 1 from } w \]
Clocks and infinite binary words

buffer

\[ \text{size}(w_1, w_2) = \max_{i \in \mathbb{N}} (O_{w_1}(i) - O_{w_2}(i)) \]

sub-typing

\[ w_1 <: w_2 \iff \exists n \in \mathbb{N}, \forall i, 0 \leq O_{w_1}(i) - O_{w_2}(i) \leq n \]
buffer \[ \text{size}(w_1, w_2) = \max_{i \in \mathbb{N}} (O_{w_1}(i) - O_{w_2}(i)) \]

sub-typing \[ w_1 <: w_2 \overset{\text{def}}{\iff} \exists n \in \mathbb{N}, \forall i, 0 \leq O_{w_1}(i) - O_{w_2}(i) \leq n \]

synchronizability \[ w_1 \bowtie w_2 \overset{\text{def}}{\iff} \exists b_1, b_2 \in \mathbb{Z}, \forall i, b_1 \leq O_{w_1}(i) - O_{w_2}(i) \leq b_2 \]

precedence \[ w_1 \preceq w_2 \overset{\text{def}}{\iff} \forall i, O_{w_1}(i) \geq O_{w_2}(i) \]
**Multi-clock**

\[
\begin{align*}
c & ::= \ w \mid c \text{ on } w \quad w \in (0 + 1) \omega
\end{align*}
\]

\(c \text{ on } w\) is a **sub-clock** of \(c\), by moving in \(w\) at the pace of \(c\). E.g.,
\(1(10) \text{ on } (01) = (0100)\).

| \text{base} \quad | 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots & (1) |
|-----------------|
| \text{\text{base on } } p_1 \quad | 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \ldots & 1(10) |
| \text{p}_1 \quad | 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \ldots & 1(10) |
| \text{base on } p_1 \quad | 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \ldots & 1(10) |
| \text{p}_2 \quad | 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \ldots & (01) |
| \text{base on } p_1 \text{ on } p_2 \quad | 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \ldots & (0100) |

For ultimately periodic clocks, precedence, synchronizability and equality are decidable (but expensive)
Come-back to the language

Pure synchrony:

- close to an ML type system (e.g., SCADE 6)
- structural equality of clocks

\[
\begin{align*}
H \vdash e_1 : ck & \quad H \vdash e_2 : ck \\
\hline
H \vdash \text{op}(e_1, e_2) : ck
\end{align*}
\]

Relaxed Synchrony:

- we add a sub-typing rule:

\[
\begin{align*}
H \vdash e : ck \text{ on } w & \quad w <: w' \\
\text{(SUB)} & \\
H \vdash e : ck \text{ on } w'
\end{align*}
\]

- defines synchronization points when a buffer is inserted
What about non periodic systems?

- The same idea: synchrony + properties between clocks. Insuring the absence of deadlocks and bounded buffering.

- The exact computation with periodic clocks does not work in practice (and is useless). E.g., \((10100100)\) on \(0^{3600}(1)\) on \((101001001)\) = \(0^{9600}(10^410^710^710^2)\)

Motivations:

1. To treat long periodic patterns. To avoid an exact computation.
2. To deal with almost periodic clocks. E.g., \(\alpha\) on \(w\) where \(w = 00.( (10) + (01) )^*\)
   
   (e.g. \(w = 00\ 01\ 10\ 01\ 01\ 10\ 01\ 10\ldots\))

**Idea**: manipulate sets of clocks; turn questions into arithmetic ones
A word \( w \) can be abstracted by two lines: \( \text{abs}(w) = \langle b^0, b^1 \rangle (r) \)

\[
\text{concr} \left( \langle b^0, b^1 \rangle (r) \right) \overset{\text{def}}{\iff} \left\{ \begin{array}{l}
w[i] = 1 \implies O_w(i) \leq r \times i + b^1 \\
w[i] = 0 \implies O_w(i) \geq r \times i + b^0 \end{array} \right.
\]
Abstraction of Infinite Binary Words

Instants

Number of ones

\[ a_4 = \left\langle 3, \frac{14}{3} \right\rangle \left( \frac{1}{3} \right) \]

\[ a_5 = \left\langle -\frac{14}{3}, -3 \right\rangle \left( \frac{2}{3} \right) \]

WG2.8 meeting 29/38
Abstract Clocks as Automata

- Set of states \( \{(i, j) \in \mathbb{N}^2\} \): coordinates in the 2D-chronogram

- Finite number of state equivalence classes

- Transition function \( \delta : \begin{cases} 
\delta(1, (i, j)) = nf(i + 1, j + 1) & \text{if } j + 1 \leq r \times i + b^1 \\
\delta(0, (i, j)) = nf(i + 1, j + 0) & \text{if } j + 0 \geq r \times i + b^0 
\end{cases} \)

- Allows to check/generate clocks
Abstract Relations

Synchronizability: \( r_1 = r_2 \iff \langle b^0_1, b^1_1 \rangle (r_1) \bowtie \sim \langle b^0_2, b^1_2 \rangle (r_2) \)

Precedence: \( b^1_2 - b^0_1 < 1 \Rightarrow \langle b^0_1, b^1_1 \rangle (r) \preceq \sim \langle b^0_2, b^1_2 \rangle (r) \)

Subtyping: \( a_1 <: \sim a_2 \iff a_1 \bowtie \sim a_2 \land a_1 \preceq \sim a_2 \)

▷ proposition: \( \text{abs}(w_1) <: \sim \text{abs}(w_2) \Rightarrow w_1 <: w_2 \)

▷ buffer: \( \text{size}(a_1, a_2) = \lfloor b^1_1 - b^0_2 \rfloor \)
Abstract Operators

Composed clocks: \( c ::= w \mid \text{not } w \mid c \text{ on } c \)

Abstraction of a composed clock:

\[
\begin{align*}
\text{abs}(\text{not } w) &= \text{not}^\sim \text{abs}(w) \\
\text{abs}(c_1 \text{ on } c_2) &= \text{abs}(c_1) \text{ on}^\sim \text{abs}(c_2)
\end{align*}
\]

Operators correctness property:

\[
\begin{align*}
\text{not } w &\in \text{concr}(\text{not}^\sim \text{abs}(w)) \\
c_1 \text{ on } c_2 &\in \text{concr}(\text{abs}(c_1) \text{ on}^\sim \text{abs}(c_2))
\end{align*}
\]
Abstract Operators

\[ a_4 = \left\langle b^0, b^1 \right\rangle \left( r^3 \frac{1}{3} \right) \]
\[ a_5 = \left\langle -b^1, -b^0 \right\rangle \left( 1 - r^3 \frac{2}{3} \right) \]

\textit{not}^\sim \text{ operator definition :}

\[ \textit{not}^\sim \left\langle b^0, b^1 \right\rangle (r) = \left\langle -b^1, -b^0 \right\rangle (1 - r) \]
Abstract Operators

\[ a_1 \sim a_2 = \langle \frac{1}{5}, \frac{7}{5} \rangle \left( \frac{3}{5} \right) \sim \langle -\frac{6}{5}, -\frac{2}{5} \rangle \left( \frac{3}{5} \right) \]

\( \sim \) operator definition:

\[
\langle b^0_1 , b^1_1 \rangle (r_1 )
\]

\[
\langle b^0_2 , b^1_2 \rangle (r_2 )
\]

\[
= \langle b^0_1 \times r_2 + b^0_2 , b^1_1 \times r_2 + b^1_2 \rangle (r_1 \times r_2 )
\]

with \( b^0_1 \leq 0, \ b^0_2 \leq 0 \)
Modeling Jitter

- set of clock of rate $r = \frac{1}{3}$ and jitter 1 can be specified by $\langle -\frac{1}{3}, \frac{3}{3} \rangle \left( \frac{1}{3} \right)$
- $\langle -\frac{1}{3}, \frac{3}{3} \rangle \left( \frac{1}{3} \right) = \langle -1, 1 \rangle \left(1\right) \text{ on } \sim \langle 0, \frac{2}{3} \rangle \left( \frac{1}{3} \right)$
- $f :: \forall \alpha. \alpha \rightarrow \alpha \text{ on } \sim \langle -\frac{1}{3}, \frac{3}{3} \rangle \left( \frac{1}{3} \right)$
Most of the properties have been proved in Coq

- example of property
  
  Property on_absh_correctness:
  
  \[
  \forall (w1 : \text{ibw}) \ (w2 : \text{ibw}), \\
  \forall (a1 : \text{abstractionh}) \ (a2 : \text{abstractionh}), \\
  \forall \ H_{wf\ a1} : \text{well\_formed\_abstractionh}\ a1, \\
  \forall \ H_{wf\ a2} : \text{well\_formed\_abstractionh}\ a2, \\
  \forall \ H_{a1\ eq\ absh\ w1} : \text{in\_abstractionh}\ w1\ a1, \\
  \forall \ H_{a2\ eq\ absh\ w2} : \text{in\_abstractionh}\ w2\ a2, \\
  \text{in\_abstractionh}\ (\text{on}\ w1\ w2)\ (\text{on\_absh}\ a1\ a2).
  \]

- number of Source Lines of Code
  
  - specifications : about 1600 SLOC
  - proofs : about 5000 SLOC
abstraction of downscaler output:

$$\text{abs}((10100100) \text{ on } 0^{3600}(1) \text{ on } (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720}))$$

$$= \langle 0, \frac{7}{8}\rangle (\frac{3}{8}) \text{ on } \sim \langle -3600, -3600 \rangle (1) \text{ on } \sim \langle -400, 480 \rangle (\frac{4}{9}) = \langle -2000, -\frac{20153}{18} \rangle (\frac{1}{6})$$

minimal delay and buffer:

<table>
<thead>
<tr>
<th></th>
<th>delay</th>
<th>buffer size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>exact result</strong></td>
<td>9 598 ($\approx$ time to receive 5 HD lines)</td>
<td>192 240 ($\approx$ 267 SD lines)</td>
</tr>
<tr>
<td><strong>abstract result</strong></td>
<td>11 995 ($\approx$ time to receive 6 HD lines)</td>
<td>193 079 ($\approx$ 268 SD lines)</td>
</tr>
</tbody>
</table>
Conclusion

Ensuring synchronous and other static properties

- specify/check logical time as special types
- initially a dependent type system; now an ML type system with extension by “Laufer & Odersky”
- this is the way it is done in the Lucid Synchrone compiler the one of SCADE 6
- some other properties can be expressed as dedicated type-systems (correct initialization of registers, causality analysis)

DSL embedding

- achieving the same result by designing a DSL (e.g., in Haskell) is difficult
- how to ensure synchrony, the absence of causality loops, unbounded FIFOs (unless we forbid non-length preserving functions)?
- compilation through maximal static expansion does not work well when targeting software code